4. Task Management System

1. Understand Linked Lists

Singly Linked List (SLL):

- **Structure:** Each node contains a data part and a reference (or link) to the next node in the sequence.
- **Traversal:** Can be traversed in one direction, from the head node to the end node.
- **Operations:**
  - Insertions and deletions are efficient if done at the head or with a known node reference.
  - Searching requires O(n) time complexity as you may need to traverse the entire list.

Insertion:

- At the beginning: O(1)
- At the end: O(n)
- At a specific position: O(n)

Deletion:

- At the beginning: O(1)
- At the end: O(n)
- At a specific position: O(n)

Traversal: O(n)

Searching: O(n)

Advantages:

- Dynamic size, easy to grow and shrink.
- Efficient insertions and deletions compared to arrays.

Disadvantages:

- Sequential access only, O(n) time complexity for searching.
- Extra memory for storing references.

Doubly Linked List (DLL):

- **Structure:** Each node contains a data part and two references: one to the next node and one to the previous node.

- **Traversal:** Can be traversed in both directions, forward and backward.
- **Operations:**
  - Insertions and deletions are efficient as each node has references to both previous and next nodes.
  - Searching is similar to SLL with O(n) time complexity.

Operations:

- **Insertion:**
  - At the beginning: O(1)
  - At the end: O(n)
  - At a specific position: O(n)
- **Deletion:**
  - At the beginning: O(1)
  - At the end: O(n)
  - At a specific position: O(n)
- **Traversal:** O(n) in both directions
- **Searching:** O(n)

Advantages:

- Can be traversed in both directions.
- Easier to delete a node when a reference to it is given, as there is no need to traverse to find the previous node.

Disadvantages:

- Extra memory for storing two references per node.

<u>**Key Differences Between SLL and DLL:**</u>

1. **Memory Use:**
   - SLL: Uses less memory since each node has only one reference.
   - DLL: Uses more memory due to two references per node.
2. **Traversal:**
   - SLL: Can only traverse forward.
   - DLL: Can traverse both forward and backward.

3. **Insertion/Deletion:**
    - SLL: Easier at the beginning but more complex (O(n)) for nodes other than the head.
    - DLL: Easier for insertion/deletion of specific nodes as each node has references to both neighbors.
4. **Applications:**
    - **SLL:** Useful for simple, singly navigable lists such as implementing stacks, adjacency lists in graphs.
    - **DLL:** Useful for more complex data structures needing bi-directional traversal such as navigation systems, undo-redo functionality.

2.Setup:

```
class Task {
    private int taskId;
    private String taskName;
    private String status;

    public Task(int taskId, String taskName, String status) {
        this.taskId = taskId;
        this.taskName = taskName;
        this.status = status;
    }

    public int getTaskId() {
        return taskId;
    }

    public String getTaskName() {
        return taskName;
    }

    public String getStatus() {
```

```
        return status;
    }

    @Override
    public String toString() {
        return "Task ID: " + taskId + ", Task Name: " + taskName + ", Status: " + status;
    }
}
```

3.Implementation

```
class Node {
    Task task;
    Node next;

    public Node(Task task) {
        this.task = task;
        this.next = null;
    }
}

class TaskLinkedList {
    private Node head;

    public void addTask(Task task) {
        Node newNode = new Node(task);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
```

```
        }
        current.next = newNode;
    }
}

public Task searchTask(int taskId) {
    Node current = head;
    while (current != null) {
        if (current.task.getTaskId() == taskId) {
            return current.task;
        }
        current = current.next;
    }
    return null;
}



public void traverseTasks() {
    Node current = head;
    while (current != null) {
        System.out.println(current.task);
        current = current.next;
    }
}

public boolean deleteTask(int taskId) {
    if (head == null) {
        return false;
    }
```

```
        if (head.task.getTaskId() == taskId) {
            head = head.next;
            return true;
        }
        Node current = head;
        while (current.next != null) {
            if (current.next.task.getTaskId() == taskId) {
                current.next = current.next.next; // Bypass the node to delete
                return true;
            }
            current = current.next;
        }
        return false;
    }
}

public class TaskManagementSystem {
    public static void main(String[] args) {
        TaskLinkedList taskList = new TaskLinkedList();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("\nTask Management System");
            System.out.println("1. Add Task");
            System.out.println("2. Search Task");
            System.out.println("3. Traverse Tasks");
            System.out.println("4. Delete Task");
            System.out.println("5. Exit");
            System.out.print("Choose an option: ");
            int choice = scanner.nextInt();
```

```
switch (choice) {
    case 1:
        System.out.print("Enter Task ID: ");
        int taskId = scanner.nextInt();
        scanner.nextLine(); // Consume newline
        System.out.print("Enter Task Name: ");
        String taskName = scanner.nextLine();
        System.out.print("Enter Task Status: ");
        String status = scanner.nextLine();
        Task newTask = new Task(taskId, taskName, status);
        taskList.addTask(newTask);
        System.out.println("Task added successfully.");
        break;

    case 2:
        System.out.print("Enter Task ID to search: ");
        taskId = scanner.nextInt();
        Task foundTask = taskList.searchTask(taskId);
        if (foundTask != null) {
            System.out.println("Found Task: " + foundTask);
        } else {
            System.out.println("Task not found.");
        }
        break;

    case 3:
        System.out.println("Current Tasks:");
        taskList.traverseTasks();
        break;
```

```
            case 4:
                System.out.print("Enter Task ID to delete: ");
                taskId = scanner.nextInt();
                if (taskList.deleteTask(taskId)) {
                    System.out.println("Task deleted successfully.");
                } else {
                    System.out.println("Task not found.");
                }
                break;

            case 5:
                System.out.println("Exiting...");
                scanner.close();
                return;

            default:
                System.out.println("Invalid option. Please try again.");
        }
    }
  }
}
```

<u>Analysis</u>

Time Complexity:

- **Add Task:** O(n) in the worst case (adding to the end of the list).
- **Search Task:** O(n) as each node may need to be checked.
- **Traverse Tasks:** O(n) as each node is visited once.
- **Delete Task:** O(n) in the worst case (when the task to delete is at the end or not present).

Advantages of Linked Lists over Arrays for Dynamic Data:

- **Dynamic Size:** Linked lists can grow and shrink dynamically without the need for resizing or reallocating memory.
- **Efficient Insertions/Deletions:** Insertions and deletions are more efficient than in arrays, particularly at the head or with known references, as no shifting of elements is required.
- **Memory Usage:** Memory is allocated for each element only when needed, avoiding potential wastage of memory associated with pre-allocated array sizes.

However, linked lists have some drawbacks:

- **Memory Overhead:** Each node requires additional memory for storing references.
- **Access Time:** Accessing elements takes O(n) time compared to O(1) in arrays due to the need to traverse the list.